

Leveraging Rust Types for Program Synthesis: Appendix

JONAS FIALA, Department of Computer Science, ETH Zurich, Switzerland

SHACHAR ITZHAKY, Technion, Israel

PETER MÜLLER, Department of Computer Science, ETH Zurich, Switzerland

NADIA POLIKARPOVA, University of California, San Diego, USA

ILYA SERGEY, National University of Singapore, Singapore

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: program synthesis, program logic, Rust, type systems

ACM Reference Format:

Jonas Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2018. Leveraging Rust Types for Program Synthesis: Appendix. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 6 pages.

A SYNTHESIS RULES

This appendix contains the full set of synthesis rules: Fig. 1 lists ownership rules, Fig. 2 lists borrowing rules, and Fig. 3 lists structural rules.

Ownership Rules. The rules DROP, RENAME, and UNREACHABLE are straightforward: once a variable is dropped or renamed (moved), it is no longer accessible due to ownership-types semantics. The UNREACHABLE rule may be encountered if a specification is inconsistent or, more importantly, in combination with DEST_R when synthesizing a **match** expression one of whose branches is unsatisfiable due to a contradiction with known assumptions (e.g. the Nil case when the matched list is known to be nonempty). This rule can only ever fire if a precondition is specified; otherwise, with no assumptions, it is impossible to deduce false.

The rules PRIMITIVE, CONSTR and DEST_R give the semantics of value construction and destructuring (pattern matching) for Rust data types, for the case where the given variable has an owned type. The rules take care to update the snapshots accordingly; in DEST_R the input binding is unconstrained so that it can be applied to arbitrary arguments—by applying the rule we learn facts about it the snapshot value. On the other hand, PRIMITIVE and CONSTR force a particular choice of snapshot on the right-hand-side since they create result objects where this value can be picked as required (when applied backwards).

Note that there is no construction rule for generic types τ and that CONSTR cannot be applied to empty types (enums with no variants), since neither is constructable in Rust. While the DEST_R rule still cannot be applied to generics but is equivalent to UNREACHABLE for empty types.

Borrowing References. The borrowing rules were discussed in detail in Section 3, apart from BORROW_{IN} and BORROW_{OUT}. These rules are analogous to DEST_RBORROW and CONSTRBORROW, respectively, but when the referent is itself a reference instead of a data type (that is, these rules handle nested references).

Authors' addresses: Jonas Fiala, Department of Computer Science, ETH Zurich, Switzerland, jonas.fiala@inf.ethz.ch; Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il; Peter Müller, Department of Computer Science, ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch; Nadia Polikarpova, University of California, San Diego, USA, nadia.polikarpova@ucsd.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

2018. 2475-1421/2018/1-ART1 \$15.00

<https://doi.org/>

$\text{DROP} \frac{T \text{ not ref}}{\{x: T(v)\} \text{ drop!}(x) \{ \text{emp} \}}$	$\text{PRIMITIVE} \frac{T_a, \dots, T_x \text{ are primitive}}{\{ \phi \mid a: T_a(v) * \dots \} \text{ let } x = a \oplus \dots \left\{ \begin{array}{l} w = v \oplus \dots \wedge \phi \mid \\ x: T_x(w) * a: T_a(v) * \dots \end{array} \right\}}$
$\text{RENAME} \frac{T \text{ not ref}}{\{x: T(v)\} \text{ let } y = x \{y: T(v)\}}$	$\text{CONSTR} \frac{T = (T_0 \times \dots)^{V_0} + \dots + (T_n \times \dots)^{V_n} + \dots}{\{y: T_n(v) * \dots\} \text{ let } x = V_n(y, \dots) \{x: T(\{\delta: V_n, y: v, \dots\})\}}$
$\text{UNREACHABLE} \frac{}{\{ \text{false} \mid P \} \text{ unreachable!}() \{Q\}}$	
$\text{DESTR} \frac{T = (T_y \times \dots)^{V_0} + (T_z \times \dots)^{V_1} + \dots \quad \begin{array}{l} \{v = \{\delta: V_0, y: u, \dots\} \wedge \phi \mid y: T_y(u) * \dots * P\} c_0 \{Q\} \\ \{r, \dots\} = \text{returns}(Q) \quad \{v = \{\delta: V_1, z: w, \dots\} \wedge \phi \mid z: T_z(w) * \dots * P\} c_1 \{Q\} \quad \dots \end{array}}{\{ \phi \mid x: T(v) * P \} \text{ let } (r, \dots) = \text{match } x \{ \begin{array}{l} V_0(y, \dots) => c_0, \\ V_1(z, \dots) => c_1, \dots \end{array} \} \{Q\}}$	

Fig. 1. Ownership Rules.

Structural Rules. The rules FRAME, CONSEQUENCE, and SEQ are familiar from SL and Hoare logic. One difference about our FRAME rule is that it does not require a side condition that free variables of R are not modified by c , because in SOL a program variable can only appear in an assertion once. The CALL is similar to the one in SuSLik, except that it is applied backwards.

DROPREF

$$\frac{}{\{\phi \mid x^\theta \mapsto T(v, \hat{x})\} \text{ drop! } (x) \{v = \hat{x} \wedge \phi \mid \text{emp}\}}$$

REBORROW

$$\frac{'a: 'b \quad \mu = \text{mut} \ ? \ \theta = \{'b\} : v = \text{imm}}{\left\{ x \xrightarrow{'a}_\mu T(v, \hat{x}) \right\} \text{ let } y = \&v * x \left\{ \begin{array}{l} y \xrightarrow{'b}_v T(v, \hat{y}) * \\ x^\theta \xrightarrow{'a}_\mu T(\hat{y}, \hat{x}) \end{array} \right\}}$$

DESTRBORROW

$$\frac{\begin{array}{l} T = (T_y \times \dots)^{V_0} + (T_z \times \dots)^{V_1} + \dots \quad \{r, \dots\} = \text{returns}(Q) \quad 'b \text{ is fresh} \\ \left\{ v = \{\delta: V_0, \theta: u, \dots\} \wedge \phi \mid x^{'b} \xrightarrow{'a} T(\{\delta: V_0, \theta: \hat{y}, \dots\}, \hat{x}) * y \xrightarrow{'b} T_y(u, \hat{y}) * \dots * P \right\} c_0 \{Q\} \\ \left\{ v = \{\delta: V_1, \theta: w, \dots\} \wedge \phi \mid x^{'b} \xrightarrow{'a} T(\{\delta: V_1, \theta: \hat{z}, \dots\}, \hat{x}) * z \xrightarrow{'b} T_z(w, \hat{z}) * \dots * P \right\} c_1 \{Q\} \quad \dots \end{array}}{\left\{ \phi \mid x \xrightarrow{'a} T(v, \hat{x}) * P \right\} \text{ let } (r, \dots) = \text{match } x \left\{ \begin{array}{l} V_0(y, \dots) => c_0, \\ V_1(z, \dots) => c_1, \dots \end{array} \right\} \{Q\}}$$

CONSTRBORROW

$$\frac{T = (T_0 \times \dots)^{V_0} + (T_1 \times \dots)^{V_1} + \dots \quad \theta = \beta \cup \dots}{\left\{ x^{'b} \xrightarrow{'a} T(\{\delta: V_n, \theta: \hat{y}, \dots\}, \hat{x}) * y^\beta \xrightarrow{'b} T_n(v, \hat{y}) * \dots \right\} \text{ drop! } (y); \dots \left\{ x^\theta \mapsto T(\{\delta: V_n, \theta: v, \dots\}, \hat{x}) \right\}}$$

BORROWIN

$$\frac{}{\left\{ x \xrightarrow{'a}_\mu \&\text{mut } T((v, \hat{a}), \hat{x}) \right\} \text{ let } y = \&\mu * x \left\{ x^{'b} \xrightarrow{'a}_\mu \&\text{mut } T((\hat{y}, \hat{a}), \hat{x}) * y \xrightarrow{'b}_\mu T(v, \hat{y}) \right\}}$$

BORROWOUT

$$\frac{}{\left\{ x^{'b} \xrightarrow{'a} \&\text{mut } T((\hat{y}, \hat{a}), \hat{x}) * y^\theta \xrightarrow{'b} T(v, \hat{y}) \right\} \text{ drop! } (y) \left\{ x^\theta \xrightarrow{'a} \&\text{mut } T((v, \hat{a}), \hat{x}) \right\}}$$

Fig. 2. Borrowing Rules.

<div style="text-align: center;">FRAME</div> $\frac{\{\phi \mid P\} \ c \ \{\psi \mid Q\}}{\{\phi \mid P * R\} \ c \ \{\psi \mid Q * R\}}$ <div style="text-align: center;">EMP</div> $\frac{\phi \implies \psi}{\{\phi \mid \text{emp}\} \ c \ \{\psi \mid \text{emp}\}}$	<div style="text-align: center;">CONSEQUENCE</div> $\frac{\phi \implies \phi' \quad \psi' \implies \psi}{\{\phi' \mid P\} \ c \ \{\psi' \mid Q\}} \quad \frac{}{\{\phi \mid P\} \ c \ \{\psi \mid Q\}}$ <div style="text-align: center;">CALL</div> $\frac{\phi \implies \phi' \quad \{\phi' \mid a: T_a(v) * \dots\} \ \text{foo } \{\psi \mid \text{result}: T(w)\}}{\{\phi \mid a: T_a(v) * \dots * P\} \ c; \text{ let } r = \text{foo}(a, \dots) \ \{\phi \wedge \psi \mid r: T(w) * P\}}$	<div style="text-align: center;">SEQ</div> $\frac{\{\phi \mid P\} \ c_0 \ \{\psi \mid Q\} \quad \{\psi \mid Q\} \ c_1 \ \{\chi \mid R\}}{\{\phi \mid P\} \ c_0; c_1 \ \{\chi \mid R\}}$
--	---	---

Fig. 3. Structural rules.

B NOTABLE SYNTHESIS RESULTS

B.1 Synthesis from Types Alone

Below is a sample of interesting examples from the “non-primitive” part of the 100-CRATES test suite; these programs are synthesized just from their type signatures.

```

// h2/src/frame/data.rs:28 (identical to actual impl)
fn new(stream_id: StreamId, payload: T) -> Data<T> {
    let flags = DataFlags(0);
    Data { stream_id, data: payload, flags, pad_len: None }
}

// nom/src/internal.rs:348 (close to actual impl, but Ok case uses closure)
fn parse(&mut self, i: I) -> Result<I, O2>, Err<E>> {
    let result = self.f.parse(i);
    match result {
        // Should be:
        // Result::Ok((i, o)) => Result::Ok((i, (self.g(o))),
        Result::Ok(_) => {
            let _0 = Err::Incomplete(Needed::Unknown);
            Result::Err(_0)
        }
        Result::Err(_0) => Result::Err(_0),
    }
}

// either/src/lib.rs:832 (identical to actual impl)
fn factor_first(self) -> (T, Either<A, B>) {
    match self {
        Either::Left(_0) => {
            let _1 = Either::Left(_0.1);
            (_0.0, _1)
        }
        Either::Right(_0) => {
            let _1 = Either::Right(_0.1);
            (_0.0, _1)
        }
    }
}

// syn/src/punctuated.rs:984 (identical to actual impl)
fn into_tuple(self) -> (T, Option<P>) {
    match self {
        Pair::Punctuated(_0, _1) => {
            let _1 = Some(_1);
            (_0, _1)
        }
        Pair::End(_0) => (_0, None),
    }
}

```

B.2 Complex Reasoning about References

The next function for a linked list iterator, which operates on the `Iter` datatype with a reference-typed field.

```

// SLL Tutorial definition, rather than our simplified definition
struct List<T> { elem: T, next: Option<Box<List<T>>> }
pub struct Iter<'a, T> { next: Option<&'a List<T>> }

```

```

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    #[ensures(match self.len() {
        0 => (^self).len() == 0 && matches!(result, None),
        v => (^self).len() == v-1 && matches!(result, Some(_)),
    })]
    fn next(&mut self) -> Option<Self::Item> {
        let (new, result) = match &mut self.next {
            None => (None, None),
            Some(_0) => match &_0.next {
                None => {
                    let result = Some(&_0.elem);
                    (None, result)
                }
                Some(_0_next_de) => {
                    let result = Some(&_0.elem);
                    let new = Some(&*_0_next_de);
                    (new, result)
                }
            },
        };
        self.next = new;
        result
    }
}

```

B.3 Complex Recursion

The *non-destructive append* benchmark takes as input immutable references to two linked lists and creates a new list containing elements of both. The main function, `sll_append_copy`, recursively traverses one of the lists, and upon reaching the end, it calls the auxiliary `sll_append_copy_7`, whose task is to recursively copy the other list. This auxiliary is not provided by the user, but is automatically discovered by RusSOL.

```

// Simple linked list
enum List<T> { Nil, Cons(Box<(T, List<T>)> ) }

#[ensures(result.len() == x1.len() + x2.len())]
fn sll_append_copy<T: Copy>(x1: &List<T>, x2: &List<T>) -> List<T> {
    match x2 {
        List::Nil => sll_append_copy_7(x1),
        List::Cons { elem, next } => {
            let de = *elem;
            let result = sll_append_copy(x1, &***next);
            let next = Box::new(result);
            List::Cons { elem: de, next }
        }
    }
}

fn sll_append_copy_7<T: Copy>(x: &List<T>) -> List<T> {
    match x {
        List::Nil => List::Nil,

```

```

List::Cons { elem, next } => {
  let de = *elem;
  let result = sll_append_copy_7(next);
  let next = Box::new(result);
  List::Cons { elem: de, next }
}
}
}

```

The *rose tree copy* benchmark takes as input an immutable reference to a rose tree and creates a new tree with the same set of elements. A rose tree is a variable-arity tree, whose children are stored in a linked list. To copy a rose tree, the synthesizer needs to create two mutually recursive functions: `copy` copies a tree and calls `copy_8` to copy its children; `copy_8` copies a list of trees and calls `copy` to copy each of them.

```

// A rose tree is either empty or a node with a value and a linked list of subtrees
enum Tree<T> { Nil, Cons { elem: T, next: List<Tree<T>> } }
// Simple linked list
enum List<T> { Nil, Cons(Box<(T, List<T>>)>) }

impl<T> Tree<T> {
  #[ensures(result.elems() == self.elems())]
  fn copy(&self) -> Tree<T> {
    match self {
      Tree::Nil => Tree::Nil,
      Tree::Cons { elem, next } => {
        let de = *elem;
        Self::copy_8(de, elem, next) // mutual recursion
      }
    }
  }
}

fn copy_8(de: T, elem_self: &T, next: &List<Tree<T>>) -> Tree<T> {
  match next {
    List::Nil => Tree::Cons { elem: de, next: List::Nil },
    List::Cons(_0) => {
      let result = _0.0.copy(); // mutual recursion
      match result {
        Tree::Nil => Self::copy_8(de, elem_self, &_0.1),
        Tree::Cons { elem: elem_result, next } => {
          let result = Self::copy_8(de, elem_self, &_0.1);
          let bx = (result, next);
          let _0 = Box::new(bx);
          let next = List::Cons(_0);
          Tree::Cons { elem: elem_result, next }
        }
      }
    }
  }
}
}
}
}
}

```